Project Threads Design

Group 43

Name	Autograder Login	Email
Sarvagya Somvanshi	student176	sarvagya@berkeley.edu
Nathan Canera	student282	nathancanera@berkeley.edu
John Chang	student271	jchang04@berkeley.edu
Shengxiang Lin	student35	linshengxiang@berkeley.edu

Efficient Alarm Clock

Data Structures and Functions

thread.h

```
struct thread {
/* other attributes omitted for brevity */
/* wake_time = timer_ticks() + ticks */
  int64_t wake_time;
}
```

thread.c

```
static struct list sleeping_list;
```

Algorithms

We know that the Pintos timer (8254 PIT) will generate a timer interrupt every TIMER_FREQ ticks – we can leverage this to make the alarm clock efficient.

Define static struct list sleeping_list in thread.c, which will store all currently sleeping threads in order of waking time; initialize sleeping_list by calling list_init(&sleeping_list) in thread_init.
Define attribute wake_time in struct thread in thread.h, which stores the time when a sleeping thread is allowed to wake up. This attribute will be calculated by calling

timer_ticks (located in timer.c) to determine the number of timer ticks since the OS booted, and adding the variable ticks passed into timer_sleep.

`timer_sleep`
The thread's wake_time should be calculated, and the thread should be placed onto sleeping_list using `list_insert_ordered`. timer_sleep should call thread_block (located in thread.c) to switch the thread's status to BLOCKED.

`timer_interrupt`
When the timer generates an interrupt, we should iterate through sleeping_list and compare the wake_time of each thread to the current time (in ticks) returned by timer_ticks(). While the thread we are currently looking at has wake_time <= current time, we call thread_unblock and place the thread onto the fifo_ready_list (defined in thread.c). Once we find a thread that has wake_time > current_time, we stop iterating, since the list is ordered by wake_time.

Synchronization

No synchronization is necessary as only the kernel will access sleeping_list.

Rationale

The runtime of iterating through `sleeping_list` is O(N) in the worst case, since we expect to prematurely stop iterating once we encounter a thread that should not be woken up yet.
We considered checking through `sleeping_list` every K interrupts, instead of every interrupt. In this case a variable interrupt_counter would be maintained within timer_interrupt. However, we cannot determine a good value for K without coding out the efficient alarm clock first.

Strict Priority Scheduler

Data Structures and Functions

We added `base_priority` to the `thread` structure to store the priority given to the thread when it was created, and used its original `priority` to store the current running priority of the thread.

```
struct thread {
 /*REST OMITTED*/
  int base_priority; // Original priority
  int effective_priority; // Current priority after donations
  struct lock *donated_to; // Lock or thread this thread donat
es priority to, if any
  list donated_locks; // List of locks receiving priority dona
tions from this thread
}
```

The `donated_locks` list keeps track of locks receiving priority donations from this thread, facilitating efficient priority tracking and updates. Meanwhile the `donated_to` points to the lock that this thread has donated priority to.

We also maintain a priority queue which keeps track of all threads waiting to be run, sorted from highest priority to lowest.

```
list* priority_queue[64]; //Sorted list to keep track of all t
hreads
```

Algorithms

Strict Priority Scheduling

`thread_schedule`

This is where the main scheduler logic will be implemented. The scheduler is modified to always select the thread the highest `effective_priority` from the `priority_queue` , ensuring strict priority ordering. When multiple threads come in with the same priority, we can utilize a round-robin approach to give equal runtime to threads with the equal priority by rating them around after a set time period. We could keep the priority list sorted by sorting in the case of every addition. For every removal, the priority list will remain sorted.

Signaling

`sema_up`
When we call upon `sema_up`, we wake up the highest-priority thread waiting on the semaphore by selecting the thread with the highest `effective_priority` in the semaphore's waiting list. If this newly woken thread has a higher priority than the current running thread, the current thread calls `yield_thread` to yields the CPU, allowing this newly awoken highest-priority thread to run.

`cond_signal`
Similarly, when `cond_signal` is called on a condition variable, it wakes the highest-priority thread in the waiting list. If the awakened thread's priority exceeds that of the current running thread, `thread_yield` is called to allow the highest-priority thread to begin running immediately.

`thread_create`
On the instance where a new thread is instantiated, its `effective_priority` is compared with that of the current thread and in the case it is higher, the current thread would need to be paused through `thread_yield` so that the higher-priority thread can begin running immediately.

Lock Priority Donation
`lock_acquire`
When a thread attempts to gain access to a lock, it first checks if the lock is currently held by another thread with a lower `effective_priority`. If this is the case, the thread will donate its `effective_priority` to the lock holder, boosting the priority of the thread utilizing the lock to allow it to release the lock more quickly. This is accomplished by setting `donated_to` to the lock and adding the lock holder to the `donated_from` list of the requesting thread.
If the thread holding the lock is itself waiting for another lock, the donation is recursively propagated up the chain through each dependent lock and thread. This ensures that priority donation flows correctly through all dependencies, allowing the highest-priority thread to eventually obtain the lock it requires. The `effective_priority` of each thread in this dependency chain is updated accordingly to reflect any new priority donations.

`lock_release`
When a thread releases a lock, it removes all donations associated specifically with that lock, as it no longer needs to prioritize the thread it was interacting with. The

`effective_priority` of the releasing thread is then recalculated based on its original `base_priority` and any remaining active donations from other locks, using the `thread_update_priority` function. This function considers all entries in the `donated_from` list, recalculating the priority based on the highest-priority donation still active. If, after releasing the lock, the current thread's `effective_priority` is no longer the highest in the system, it will call `thread_yield`, yielding the CPU to allow the next highest-priority thread to run.

Helper Functions

`thread_update_priority`

This function begins by initializing the thread's `effective_priority` to its `base_priority`. It then iterates through all active donations listed in the thread's `donated_from` list, comparing each donor's priority. For each lock or thread in `donated_from`, the function checks if the priority of the donating thread is higher than the current `effective_priority`; if so, it updates `effective_priority` to match this highest-priority donation. After recalculating, the function checks if the thread's new `effective_priority` is lower than that of any thread currently in the `priority_queue`. If it is, the thread calls `thread_yield` to give up the CPU, allowing the next highest-priority thread to execute.

`thread_get_priority`

This function returns `current_thread()->effective_priority`.

Synchronization

In this part of the project, a key way to handle synchronization could be through through controlled interrupt handling rather than extensive use of explicit locks. Certain structures, like the scheduler `priority_queue` semaphore waiting list, and condition variable waiting list, inherently avoid race conditions due to careful management: the scheduler `priority_queue` and semaphore waiting list are only modified when interrupts are disabled, while the condition variable waiting list is protected by locks. For operations involving priority recalculations or lock acquisitions, interrupts are disabled to ensure atomic updates, preventing unexpected context switches that could disrupt priority order. Specifically, during `thread_update_priority`, disabling interrupts will keep the donation setup stable, avoiding any priority changes mid-calculation.

Rationale

By assigning tasks based on priority and enabling priority inheritance through donations, this scheduler avoids priority inversion. The use of both `base_priority` and `effective_priority` allows to managing dynamic donations, preserving the threads original priority series while allowing temporary adjustments to meet the immediate and inherent needs. The `priority_queue` is kept sorted at all times based on `effective_priority`, making sure that the scheduler can always select the highest-priority thread efficiently. This sorted order is maintained by adjusting the queue when threads are added or removed, providing constant access to highest priority task without needing to re-sort the entire list.

User Threads

Data Structures and Functions

First, we will be adding some new attributes to our beloved process struct owned by process.h.
We need to implement some new functionality to support multithreading. This includes adding a new list of threads, a list of the user's semaphores, and a list of the user's locks. We should also have the id's for the next available lock, semaphore, and tid for the sake of the threads.

For the process struct we need a pcb lock for changes to the process struct. This prevents any racing when threads are editing their parent process and maintains a nice and neat pcb.

The elements in the thread list will have the threads tid, its status, its semaphore for joining, and a pointer to its stack. The list of locks will have the id to the lock and the actual lock object.

The list of semaphores will have a similar structure with the id of the semaphore and the actual semaphore but also the tid of the joiner and joinee.

For synchronization's sake we will include a struct with a semaphore to signal to the parent thread the status of its child's creation and prevent it proceeding before its child is created.

Finally, we have included a list of freed pages to prevent fragmentation when adding new ones.

```
struct process {
  //All the new attributes with the old ones omitted.
  struct lock pcb_lock;
  struct list thread_list;
  struct semaphore user_semas[256];
  struct lock user_locks[256];
  /* locks */
  int next_tid;
  int next_sema;
  int next_lock;
  /* child creation signal */
  struct child_status c_status;
  struct condition exit_status
  /* list of freed pages to prevent fragmentation */
  struct list* freedom;
}


struct thread_elem {
  int tid;
  int status;
  struct sema_elem sema_join;
  void* stack_pointer;
  struct list_elem elem;
}
```

```
struct sema_elem {
  int sid;
  int joiner_tid;
  int joinee_tid;
  struct semaphore join;
}

struct lock_elem {
  int lid;
  struct list_elem elem;
}

struct child_status {
  semaphore done;
  int status;
}
```

Algorithms

New Algorithms

`tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg)`

To create a page, we have to used the earliest address we possibly can to prevent fragmentation. An issue we might run into deciding where to put our page is that when we close pages, sometimes we close pages that are in the middle of others and if we leave those gaps unfilled it'll cause fragmentation. To solve this we'll keep a list of what pages we've freed so that when we are created a thread and allocating a page, we can allocate the earliest freed page and prevent fragmentation. We'll need to call `palloc_get_page()` to generate a new page, representing the user

stack that will be private to each thread. Then, we need to initialize a new `thread` struct and we can utilize the `stub_fun` to help setup the thread, set up `tfun` and `arg` for the target function and argument in the thread's stack.
Moreover, we would need add the new thread to the parent's `thread_list` list in the PCB.

For synchronization, we can also utilize a semaphore to the parent thread is informed of the creation status. The parent thread is blocked until the semaphore is signaled, confirming that the thread creation succeeded or failed. This prevents race conditions where the parent might proceed before the child thread is fully initialized.

`void sys_pthread_exit(void) NO RETURN`

When `sys_pthread_exit` is called, the thread needs to be freed and exited. For this, the thread first needs to call `palloc_free_page()` to release all allocated memory such as the user stack, helping prevent memory leaks. In addition, we also need to free up the other variables such as join statuses, user locks, and semaphores, includes unlocking any user locks that the thread held and signaling any semaphores it was blocking. We can utilize `cond_broadcast` to call on all the waiting threads, waking up up and calling `sys_pthread_join(calling_thread)`. These threads will then be able to clean up and complete their own execution.

Finally, `process_exit` is then called to remove the current thread from the scheduler and switch to another thread. If this is the main thread, we must join on all active threads before `process_exit` is called.

`tid_t sys_pthread_join(tid_t tid)`
Firstly, we need to iterate through the `thread_list` to find the `thread` struct that correlates with the `tid`. If the target thread is still running, the calling thread is blocked using the semaphore available in the `thread` struct in the thread. This ensures that the calling thread waits until the target thread exits. Hence, when the thread calls `sys_pthread_exit()` and signals its semaphore, the blocked thread is unblocked and can continue execution. After the target thread has exited, the exit status of the thread is returned to the calling thread. Once the target thread has exited and the waiting thread is unblocked, we would also need to remove the join status entry from `thread_list` to ensure no memory leaks.

`bool lock_init(lock_t* lock)`
When a lock is intilaized, memory needs to be allocated for the lock and the internal state needs to be unlocked and we can achieve that by utilizing the `lock_init` from `synch.c`, where a successful call to the function would result with return of `true` while a error in calling the function would return `false`. The new lock is added to the `user_locks` list within the PCB by using the `next_lock` incrementing counter to assign a unique ID.

`bool lock_acquire(lock_t* lock)`
Firstly, we would need to check if the lock exists within `user_locks` and then if it exists, `lock_acquire` from `synch.c` would be called and we will return `true`. If the lock was not found, we would have to return `false`.

`bool lock_release(lock_t* lock)`
Firstly, we would need to check if the lock exists within `user_locks` and then if it exists, `lock_release` from `synch.c` would be called and we will return `true`. If the lock was not found, we would have to return `false`.

`bool sema_init(sema_t* sema, int val)`
For this function, memory is allocated for a new semaphore, setting its `sema_count` to the specified `val` using the `sema_init` from `synch.c` to properly initialize the semaphore. Then a unique semaphore ID may be assigned using the `next_sema` count from the PCB, and the semaphore is added to the `user_semas` list in the PCB.

`bool sema_down(sema_t* sema)`
First we need to validate if `sema` exists in `user_semas` and if it does, we would need to call `sema_down` from `synch.c` to decrement its `sema_count` and return `true`. Else we would need to return `false` as the sema does not exist. If the count is greater than 0, the calling thread would be able to proceed, else it would be blocked and have to wait.

`bool sema_up(sema_t* sema)`
First we need to validate if `sema` exists in `user_semas` and if it does, we would need to call `sema_up` from `synch.c` to increment t its `sema_count` and return `true`. Else we would need to return `false` as the sema does not exist. Then if there was any

thread waiting on the semaphore, it would be unblocked so it could use the semaphore.

`tid_t get_tid(void)`
This function returns the thread ID (TID) by calling `current_thread->tid` to return the TID of the current running thread.

Modifications

`pid_t exec(const char* file)`
When `exec` is called by either a single-threaded or multi-threaded process, it would need to create a new process with a single thread of control, designated as the main thread. This allows additional threads to be created in the new process via `pthread_create` calls after `exec` completes.

`int wait(pid_t)`
If a user thread waits on a child process, only the thread calling `wait` is suspended meaning other threads in the parent process can continue executing independently. This ensures that thread synchronization at the process level does not impact concurrent operations of other threads in the parent.

`void exit(int status)`
When `exit` is called in a multi-threaded process by the main thread, all active threads would need to be terminated. The backing kernel threads for each user thread would release all allocated resources before terminating. Threads should be woken up appropriately if waiting on others (e.g., in `pthread_join`).

Synchronization

The pcb is a shared resource. A part of being a shared resources in a multithreaded environment is worrying about race conditions. To alleviate this worry, we will include a lock for the pcb to be used whenever a user-thread is using a function that edits the pcb, `pcb_lock`. It can be initialized at the start of a process and they will likely be used at thread creation. In instances where multiple threads are created, the race conditions can happen on adjustment of stack pointers, the thread list, etc...

Next, we focus on thread creation again, this time though specifically from the perspective of a parent thread creating a child thread. To prevent our parent thread from continuing before the child thread finishes creation. we add an attribute to the process struct to keep track of the child's creation status and stop our parent thread from getting ahead of itself, `c_status`.

To prevent deadlocks, we will have to be very careful when making threads wait. This includes instances like the `pthread_join` function and `process_wait`. Before starting to wait, we will ensure that the threads calling these funcitons release all their locks to prevent any deadlocking from occurring.

Similar to our thread creation, its important that we keep track of our thread exiting to ensure that we aren't having multiple threads exiting at the same time. We'll do this with a condition variable and broadcast out to the joiners to wake them. While we wait for the rest of the threads to peacefully expire, the condition variable in our process struct ensures that none of them call process_exit and produce some unsatisfactory race conditions. Once all the rest of the threads are finished, our lone exiting thread will then exit itself and mission complete.

Rationale

By adding attributes like `pcb_lock` to control concurrent access to shared resources in the `process` struct, this design maintains data consistency and prevents race conditions, which are common in multithreading environments. The use of lists for managing threads, locks, and semaphores enables organized tracking of resources, allowing for quick access and updates with predictable runtime complexity. For instance, freeing pages upon thread exit prevents memory fragmentation and optimizes memory reuse, reducing the risk of memory leaks and enhancing memory management efficiency.

Synchronizing the parent thread during child creation with a semaphore helps prevent premature access, avoiding race conditions and enhancing reliability. Similarly, the use of condition variables to coordinate orderly thread exits ensures that resources are released systematically, preventing deadlocks. The approach to `pthread_join` and `sys_pthread_exit` also strengthens synchronization by only blocking relevant threads, thus optimizing runtime performance by reducing unnecessary thread suspensions. This structured handling of user threads ensures efficient memory and resource management and enhances system robustness.

Concept check

1. When a kernel thread in Pintos calls `thread_exit`, the page containing its stack and Thread Control Block (TCB) (`struct thread`) is freed later in `thread_schedule_tail`. We avoid calling `palloc_free_page` directly in `thread_exit` because the CPU's `%esp` register still points to this page. If we freed the memory immediately, we risk accessing freed memory, leading to undefined behavior when using stack variables or executing any stack-based operations.
2. When the thread_tick function is called by the timer interrupt handler, it executes on the kernel stack. Specifically, it runs on the kernel stack of the currently executing thread.
3. `threadA: lock_acquire(&lockA); // Successful acquisition of lockA`
   `threadB: lock_acquire(&lockB); // Successful acquisition of lockB`
   `threadA: lock_acquire(&lockB); // Waiting for lockB, blocked`
   `threadB: lock_acquire(&lockA); // Waiting for lockA, blocked`
   At this point, both threadA and threadB are blocked, so this schedule implements deadlock.
4. Killing thread B can lead to a lot of different unpredictable behavior. Thread B could be in a critical zone and executing some code that other threads will rely on in the future. Killing it without allowing it to complete could end up ruining a numerous amount of different threads with deadlocks, unreleasable locks, memory leaks, and a plethora of other undefined behaviours.
5.
   a. Create two threads, thread_high and thread_low. thread_high has a lower priority and thread_low has a higher base priority, but through priority donations, thread_low's effective priority will be lower than thread_high.
   b. thread_low acquires a mutually exclusive lock that causes it to donate its priority to thread_high.
   c. thread_low waits for a semaphore, which causes it to enter a blocking state.
   d. thread_high tries to acquire the same mutually exclusive lock, which causes it to block because thread_low holds the lock.
   e. At some point, the semaphore is released. If the semaphore's sema_up function is implemented correctly, it should unblock thread_low because thread_low has a higher effective priority than thread_high.
   f. If the semaphore's sema_up function is implemented incorrectly, it may unblock thread_high because it only considers the base priority.

Expected output:
thread_low: Locked mutex and should have donated priority to thread_high.
thread_low: semaphore_up should have woken me up due to higher effective priority.

Actual output:
thread_low: Locked mutex and should have donated priority to thread_high.
thread_high: Acquired mutex. This should not happen if sema_up is correct.
In this actual output, we see that thread_high acquires the mutex lock even though thread_low's effective priority should be higher, indicating that the semaphore's sema_up function is not handling priority donations correctly.